Sohan Lal, Ben Juurlink

A Quantitative Study of Locality in GPU Caches

Lal, Sohan; Juurlink, Ben (2020): A Quantitative Study of Locality in GPU Caches. Accepted for International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XX), July 5 – 9, 2020.

A Quantitative Study of Locality in GPU Caches

Sohan Lal and Ben Juurlink

Technische Universität Berlin, Germany

Abstract. Traditionally, GPUs only had programmer-managed caches. The advent of hardware-managed caches accelerated the use of GPUs for general-purpose computing. However, as GPU caches are shared by thousands of threads, they are usually a victim of contention and can suffer from thrashing and high miss rate, in particular, for memory-divergent workloads. As data locality is crucial for performance, there have been several efforts focusing on exploiting data locality in GPUs. However, there is a lack of quantitative analysis of data locality and data reuse in GPUs. In this paper, we quantitatively study the data locality and its limits in GPUs. We observe that data locality is much higher than exploited by current GPUs. We show that, on the one hand, the low spatial utilization of cache lines justifies the use of demand-fetched caches. On the other hand, the much higher actual spatial utilization of cache lines shows the lost spatial locality and presents opportunities for further optimizing the cache design.

Keywords: Data Locality · GPU Caches · Memory Divergence.

1 Introduction

GPUs have been very successful in accelerating general-purpose applications from different domains. The advent of hardware-managed caches further accelerated the use of GPUs for general-purpose computing by exploiting temporal and spatial locality. Caches reduce off-chip memory traffic and cut down pressure on memory bandwidth, however, using caches efficiently is a difficult task because a GPU employs a large number of threads, and GPU caches are small that lead to high contention and thrashing. In particular, there is a class of applications known as irregular applications that poorly utilize GPU caches. Irregular applications have memory divergence and/or control divergence, and they span a broad range of domains. GPUs issue concurrent memory accesses to consecutive addresses by coalescing memory accesses of threads in a warp. However, it may not always be possible to coalesce individual thread accesses due to scattered memory accesses that lead to memory divergence.

Memory divergence is known to cause many problems, including data overfetch which can waste cache capacity, consume scarce resources such as MSHRs and memory bandwidth, thus, further making it hard to utilize caches efficiently. As exploiting data locality is important for performance, there have been several studies to exploit the data locality in GPUs [15,7]. Moreover, the recent

generations of GPUs have adapted the cache design to tackle issues such as data over-fetch caused by memory divergence. For example, Maxwell, Pascal, and Volta GPU architectures use sector caches to fetch only the sectors that are requested instead of always fetching all sectors of a cache line. We show that there is a higher locality in GPU caches than currently exploited by demand-fetched caches, offering opportunities for further optimizing the cache design for higher performance. Moreover, there is a lack of quantitative analysis of data locality and data reuse in GPUs. Therefore, in this paper, we quantitatively study the data locality and its limits in GPUs at different granularities. We observe that data locality is much higher than currently exploited by GPUs.

In summary, we make the following main contributions:

- We present a comprehensive, quantitative and limit study of locality in GPU data caches for memory-divergent workloads.
- We show that about 50% of cache capacity, and other scarce resources such as NoC bandwidth, memory bandwidth are wasted due to data over-fetch.
- We show that memory-divergent workloads have much higher locality than exploited by GPUs as locality gets destroyed due to cache thrashing.
- Our analysis shows that 57% of the cache lines are never re-referenced. However, the limit study shows that actually, only 30% of the cache lines are never re-referenced and 27% are evicted before re-reference.
- We show that the low spatial utilization of cache lines justifies the design decision to fetch sectors based on demand, however, the much higher actual spatial utilization of cache lines shows the lost spatial locality and presents opportunities for further optimizing cache designs.

The paper is organized as follows. In Section 2, we briefly discuss GPU data caches and classify locality. In Section 3, we explain the experimental setup. Section 4 presents the quantitative results. Section 5 describes related work. Finally, we conclude in Section 6.

2 Background on GPU L1 and L2 Data Caches

Traditionally, GPUs only had programmer-managed caches, however, with the advent of Fermi architecture, GPUs started using hardware-managed caches. In fact, the hardware-managed caches have accelerated the use of GPUs for general-purpose computing. There are several works which compared the performance of Tesla and Fermi GPUs and reported that hardware-managed caches play an important role in the higher performance of Fermi GPUs over Tesla GPUs [18,6,3]. GPU caches face different design challenges than CPU caches due to their different characteristics. For instance, write and allocation policies are quite different from CPU caches. The L1 data cache is only write-back for local accesses and write-evict for global accesses whereas in a CPU we either have write-back or write-through caches. The allocation policy is usually no-write allocate for global accesses and write-allocate for only local accesses. The deviation in write and allocation policies is to cater to the different requirements of GPU workloads

Table 1: Summary of locality classification.

Locality Category	Description
Intra-warp	Locality from threads of the same warp
Inter-warp	Locality from different warps
Intra-CTA	Locality from warps of the same CTA
Inter-CTA	Locality from different CTAs

and smaller caches. GPU caches are shared by thousands of threads which make them a scarce resource and a victim of a lot of contention. Furthermore, due to the streaming nature of many GPU applications and smaller cache sizes, caches can suffer from thrashing and high miss rate. Therefore, exploiting locality in GPU caches is hard due to a large number of active threads and smaller caches. In fact, some works reported negative performance results with caches [7,14].

Thread-blocks or co-operative thread arrays (CTA) in NVIDIA terminology are independent units of scheduling on streaming multiprocessors (SMs) and a thread-block can be scheduled on any SM. This feature allows transparent scalability as simply more thread-blocks can be scheduled in parallel when more SMs are available and vice-versa. As thread-blocks can be scheduled on any SM, it is very hard to exploit inter-thread block locality at L1 caches because the L1 cache is private to an SM. For example, when two thread-blocks have inter-thread block locality but they are scheduled on different SMs, there is no way to exploit inter-thread block locality at L1. L2 cache is useful for exploiting inter-thread block locality in this case as L2 cache is shared among all SMs. Therefore, we have a relatively large L2 cache, which helps to filter requests to off-chip memory. L1 and L2 caches on GPUs are smaller than L1 and L2 caches in CPUs, but they have high bandwidth.

A typical cache line size is 128B in GPUs. The loads and stores were normally serviced at the granularity of a cache line until the Fermi architecture. However, starting from the Kepler architecture and subsequently, in other architectures such as Maxwell, Pascal, and Volta, the loads and stores can be serviced at 32B granularity. The 32B granularity is known as a sector. These architectures still have a data cache line size of 128B, but a cache line is divided into 4 sectors. There are byte masks, and on a miss, a cache will only fetch the 32B sectors that are requested. A full cache line is not automatically fetched, however, if all four sectors are requested, it is also possible to fetch a full cache line.

2.1 Locality Classification

As a thread block (also known as CTA), and a warp are the scheduling units in GPUs, we classify and study locality at thread block and warp level granularities. We classify the locality as *intra-warp locality* when a cache line is initially referenced by a warp and then re-referenced by the same warp. When a cache line is re-referenced by a different warp than the one initially requested the cache

Table 2: Summary of simulator configuration.

Parameter	Value	Parameter	Value
#SMs	16	Shared memory/SM	48KB
SM freq (MHz)	822	L1 $ size/SM	16KB
Max #Threads per SM	1536	L2 $ size	768KB
Max #CTA per SM	8	# Memory controllers	6
Max CTA size	512	Memory type	GDDR5
#FUs per SM	32	Memory clock	2004 MHz
#Registers/SM	32K	Memory bandwidth	192.4 GB/s

line, we classify such a locality as *inter-warp locality*. Similar to intra-warp and inter-warp locality, we classify the locality as *intra-CTA locality* and *inter-CTA locality*. Table 1 shows a summary of the locality classification.

3 Experimental Setup

In this section, we describe our experimental methodology.

3.1 Simulator

We use gpgpu-sim simulator for simulating different benchmarks [2]. We configure the simulator to have two level data caches with a cache line size of 128B. Table 2 summarizes the main configuration parameters of the simulator.

3.2 Benchmarks

Table 3 shows the benchmarks used for the experimental evaluation. We include benchmarks from the popular Rodinia benchmark suite [4] and CUDA SDK [13] that have memory divergence. From each benchmark, we select a single kernel launch with the highest memory divergence, and when a benchmark has multiple kernels, we only include kernels that have memory-divergence. A benchmark is memory divergent if all intra-warp memory accesses of a load or a store instruction cannot be coalesced into one (two) memory transaction(s) depending upon whether the coalescing is done at half (full) warp. We use coalescing efficiency to note the degree of memory divergence for each benchmark. The lower the coalescing efficiency, the higher the degree of memory divergence. The coalescing efficiency is given by the following equation:

$$\text{CE} = \sum \text{GMI} / \sum \text{GMT}$$

GMI is the total number of global memory instructions executed for a benchmark and GMT is the corresponding number of global memory transactions issued. When global memory instructions are equal to the global memory transactions, we have a case of perfect coalescing (100%), assuming 32-bit data is

Table 3: Memory-divergent benchmarks used for the experimentation. A benchmark with different subscripts designates distinct kernels of the same benchmark.

Name	Launch id	CE (%)	Description
histogram	36	3	Histograms for analysis [13]
kmeans1	1	6	k-means clustering [4]
scan1	11	5	Parallel prefix sum [13]
srad2_1	1	56	Speckle reducing anisotropic diffusion [4]
srad2_2	2	53	Speckle reducing anisotropic diffusion [4]
bfs	15	47	Breadth-first search [4]
ss1	9	11	Similarity score calculation [4]
b+tree1	1	75	Graph search [4]
b+tree2	2	75	Graph search [4]
srad1_1	5	70	Speckle reducing anisotropic diffusion [4]
srad1_2	11	79	Speckle reducing anisotropic diffusion [4]
mummergpu	1	7	Pairwise local sequence alignment [4]
storeGPU	1	41	Distributed storage systems [1]
mergesort1	2	5	Parallel merge-sort [13]
mergesort2	3	50	Parallel merge-sort [13]
mergesort3	41	67	Parallel merge-sort [13]
convSep1	1	50	Convolution [13]
convSep2	2	43	Convolution [13]
backprop1	2	64	Multi-layer perceptron training [4]
heartwall	2	54	Ultrasound image tracking [4]
hotspot	1	35	Processor temperature estimation [4]
leukocyte	3	51	Microscopy video tracking [4]

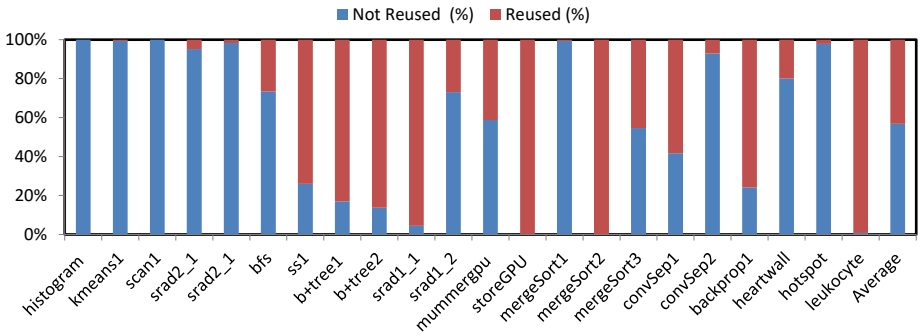
accessed by each thread. In the worst case, there can be one transaction issued per thread of a warp that corresponds to a coalescing efficiency of about 3% (1/32). Table 3 shows the coalescing efficiency of the benchmarks.

4 Results

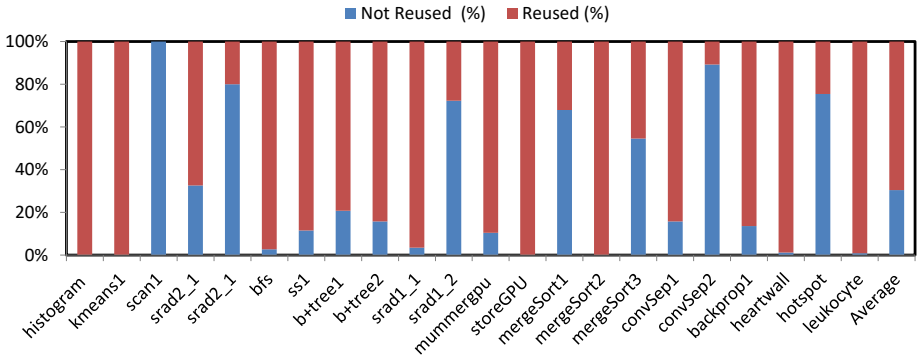
In this section, we present the results of the quantitative study of L1 and L2 data caches. In Section 4.1, we present results for cache line reuse. Section 4.2 presents the analysis of the spatial utilization of cache lines. In Section 4.3 and Section 4.4, we study locality at warp and CTA levels, respectively.

4.1 GPU Data Cache Lines Reuse Limit

Figure 1 shows the reuse of L1 data cache lines. Figure 1a shows the reuse of L1 data cache lines for 16KB size. The figure shows that for several kernels such as *histogram*, *kmeans1*, *scan1*, the cache lines are evicted without any reuse. Only a few kernels such as *storeGPU*, *mergeSort2*, and *leukocyte* have all of the cache



(a) L1 data cache lines reuse with 16KB size.



(b) Limit of L1 data cache lines reuse with with an infinite size.

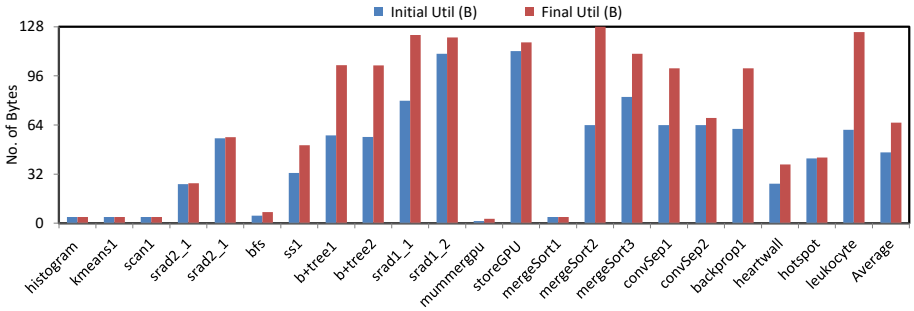
Fig. 1: GPU L1 data cache lines reuse.

lines reused. The average cache line reuse is only 43%, which means that 57% of the cache lines are never re-referenced or gets evicted before re-reference.

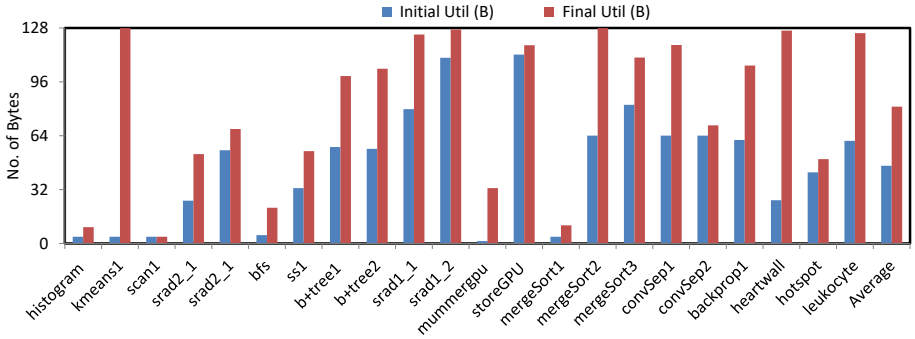
In order to study the lost opportunities to reuse cache lines due to limited cache size, we also simulate a L1 data cache with infinite size. The infinite cache size here implies that once a cache line is brought to the cache, it never gets evicted. Figure 1b shows the L1 data cache lines reuse with an infinite size. The average cache lines reuse for the infinite L1 data cache is 70%, which shows that most of the kernels have high locality but current GPUs are unable to exploit the locality due to limited cache size. However, even with the infinite cache size, 30% of the L1 data cache lines have no data reuse.

4.2 GPU Data Cache Lines Spatial Utilization Limit and Over-fetch

Figure 2 shows the spatial utilization of L1 data cache lines. Each kernel has two bars. The first bar shows the average initial utilization of a cache line, i.e., when a cache line is initially fetched. The second bar shows the average final utilization of a cache line, i.e., when a cache line is evicted. For calculating the



(a) L1 data cache lines spatial utilization with 16KB size.



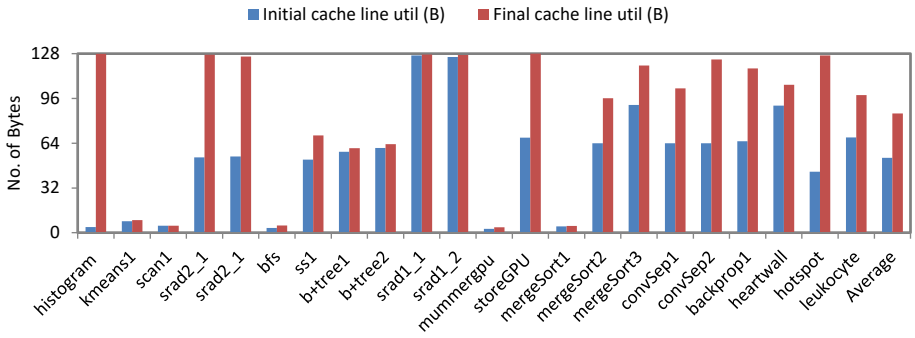
(b) Limit of L1 data cache lines spatial utilization with an infinite size.

Fig. 2: GPU L1 data cache lines spatial utilization showing data over-fetch.

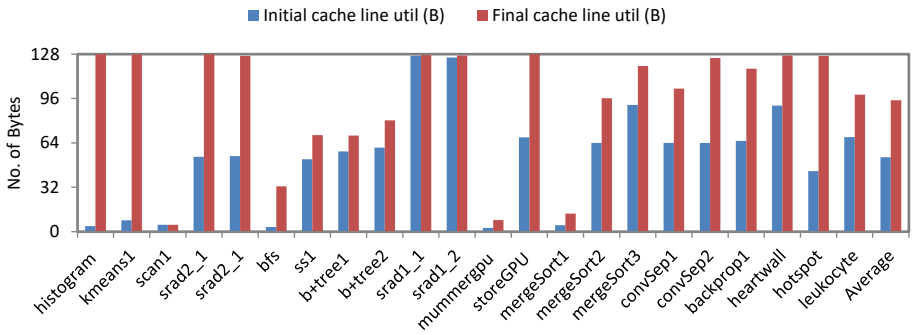
spatial utilization, we use a bit vector of length equal to a cache line size. The bits corresponding to bytes that are requested at the time of fetching a cache line are initially set and the bit vector is updated whenever a cache line is reused until the cache line is finally evicted.

Figure 2a shows the spatial utilization of the L1 data cache lines for 16KB cache size. The figure shows that for many kernels, the initial and the final utilization of cache lines is almost the same, which means there is a very low spatial utilization of L1 data cache lines. Only kernels such as *b+tree1*, *b+tree2*, *srad1*, *mergesort2*, *mergeSort3*, *convSep1*, *backprop1*, and *leukocyte* have significantly higher final utilization of cache lines. The average initial and final utilization of cache lines is 46B and 65B for all kernels. This implies that about 50% of the cache capacity, and other scarce resources such as network-on-chip bandwidth, L2 data cache bandwidth, etc., are wasted due to the data over-fetch.

The low utilization of cache lines could also be due to the limited cache capacity and high contention as a cache line may be evicted before it gets re-referenced again. To study the upper limit of the spatial utilization of cache lines, we simulate an infinite L1 data cache, which means a cache line once fetched to the cache never gets evicted. Figure 2b shows the final utilization of cache lines



(a) L2 data cache lines spatial utilization with 128KB size.

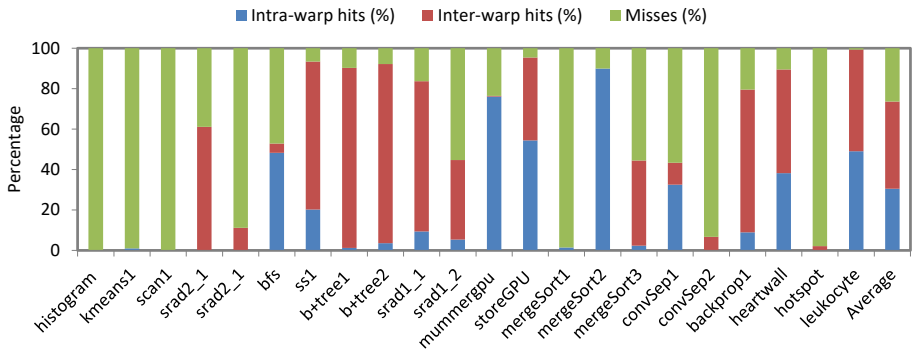


(b) Limit of L2 data cache lines spatial utilization with an infinite size.

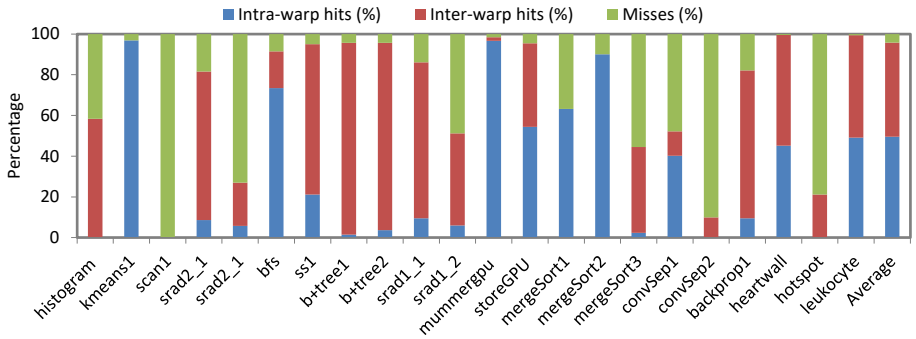
Fig. 3: GPU L2 data cache lines spatial utilization showing data over-fetch.

for an infinite L1 data cache. The figure shows that most of the kernels actually have higher spatial locality than exploited by the default cache size, however, GPUs are unable to exploit full spatial locality. The average final utilization of cache lines for the infinite L1 data cache is 81B, which is 24% higher than the default cache size. However, even with the infinite cache size, 36% of the cache capacity is still wasted. This implies that there is a potential for further improvements in the cache design.

There are two important conclusions that we can draw from the spatial utilization of cache lines. First, on the one hand, the low initial spatial utilization of cache lines justifies the sector cache design of Maxwell, Pascal, and Volta architectures. On the other hand, the much higher final spatial utilization of cache lines, as shown by our experiments, presents an opportunity for further optimizing cache designs. For example, if we only fetch the sectors on demand depending on the initial utilization of a cache line, we significantly lose the spatial locality present in the kernels. The initial sectors that need to be fetched for a memory load can be determined at the time of coalescing different memory requests of threads of a warp. If we only fetch the sectors as determined during coalescing, which is how it is done in recent GPUs, there will be significant lost



(a) Intra- and inter-warp locality of L1 data cache with 16KB size.

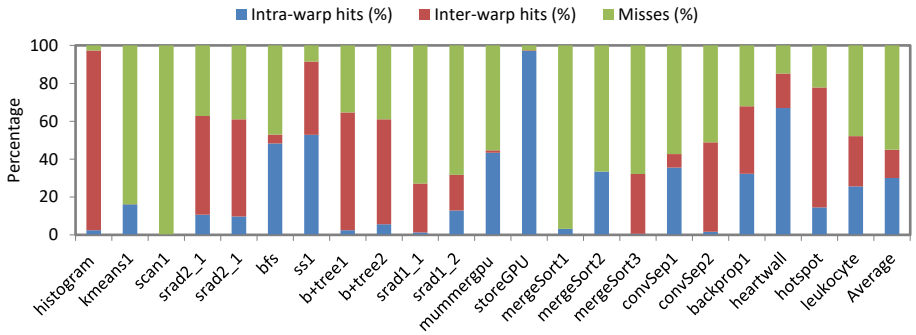


(b) Intra- and inter-warp locality of L1 data cache with an infinite size.

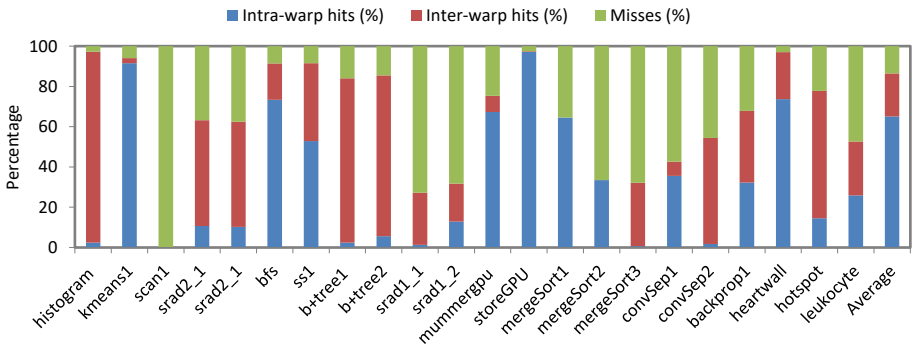
Fig. 4: Intra- and inter-warp locality of L1 data cache.

opportunities for exploiting locality as our analysis shows that the final utilization of cache lines is much higher than the initial utilization of cache lines (see Figure 2). One possible idea to exploit the lost locality is to investigate the use of spatial locality predictor for GPUs and then depending upon the prediction, fetch the predicted sectors.

We also did a similar kind of analysis for the L2 data cache. Figure 3 shows the spatial utilization of L2 data cache lines for 128KB as well as infinite cache sizes. Figure 3 shows that the L2 data cache has better spatial utilization of cache lines compared to the L1 data cache. This is because the L2 data cache is shared by multiple processors. The average initial and the final utilization of cache lines is 54B and 85B, respectively. This means that 35% of the L2 data cache capacity is wasted, which is lower than the L1 data cache but still significant to look for further optimizations. The wastage of the L2 data cache is also a measure of wasted off-chip memory bandwidth. The average final utilization of cache lines is 95B for the infinite cache, which is almost 2.0× more compared to the average initial utilization of cache lines. The big difference between the initial and the final utilization of cache lines shows that pure demand-based



(a) Intra- and inter-warp locality of L2 data cache with 128KB size.



(b) Intra- and inter-warp locality of L2 data cache with an infinite size.

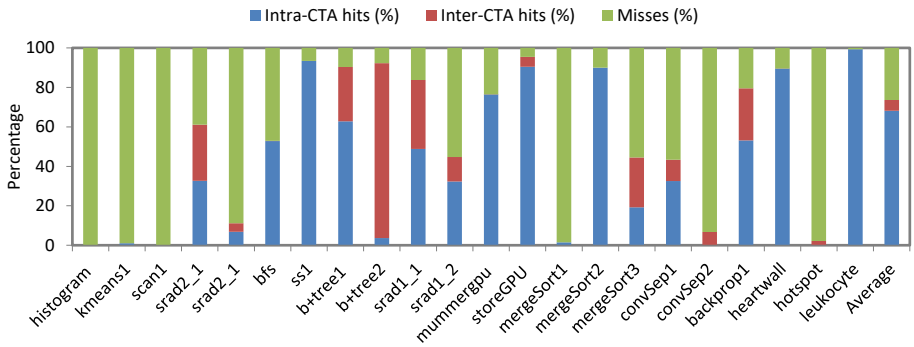
Fig. 5: Intra- and inter-warp locality of L2 data cache.

sector fetching, implemented in GPU caches, needs to be revisited to improve their performance. The analysis also shows that we need more aggressive policies to properly utilize the L1 data cache than the L2 data cache.

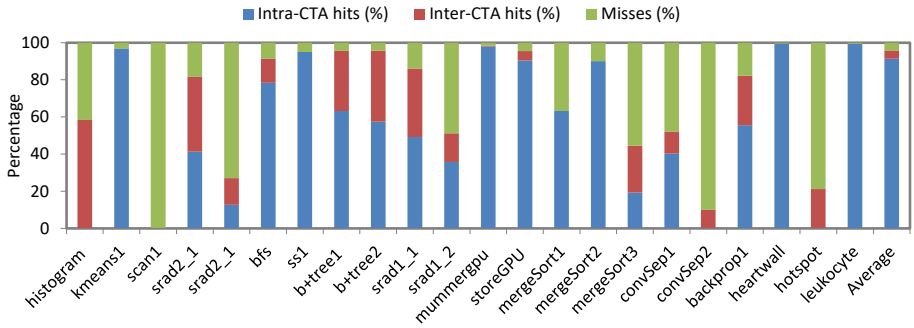
4.3 Intra- and Inter-Warp Locality

As cache hits and misses are important metrics to evaluate a cache design, we also present hits and misses for both L1 and L2 data caches. We classify hits into two categories, *intra-warp* and *inter-warp hits*. As a warp is an important scheduling unit within an streaming multiprocessor, the hit ratio at a warp granularity provides crucial information that could be used for both scheduling warps and improving cache design.

Figure 4 shows the intra- and inter-warp hits of L1 data cache. Figure 4a shows that the L1 data cache has 31% intra-warp and 43% inter-warp hits for 16KB cache size. The miss rate is 26% which is quite high. In contrast to GPUs, CPUs have a hit rate typically between 95% to 97% for a L1 cache. As discussed earlier, it is extremely difficult to exploit data locality in GPU caches due to several factors such as small size, massive multithreading.



(a) Intra- and inter-CTA locality of L1 data cache with 16KB size.

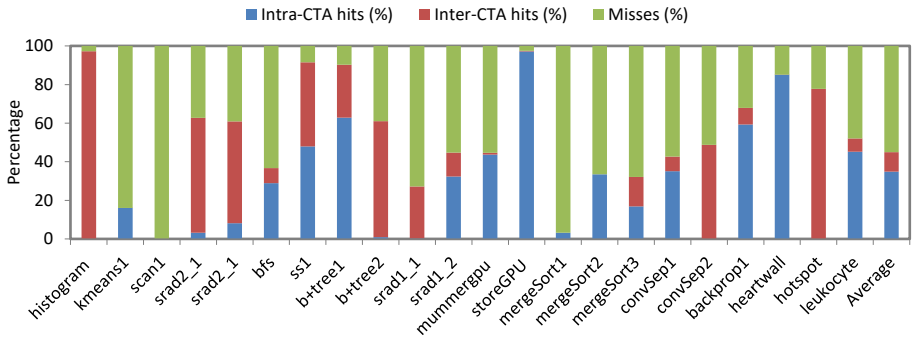


(b) Intra- and inter-CTA locality of L1 data cache with an infinite size.

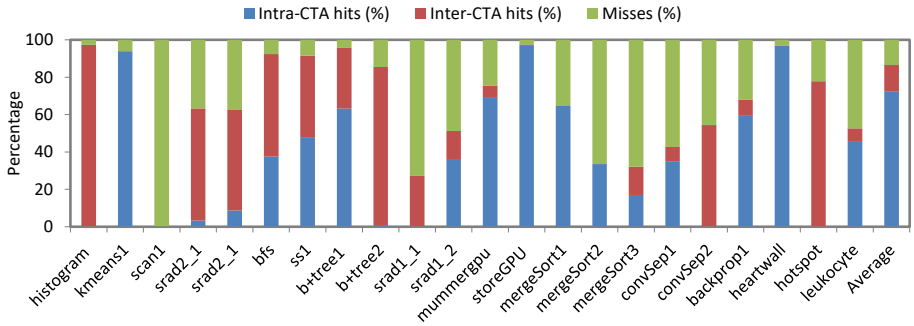
Fig. 6: Intra- and inter-CTA locality of L1 data cache.

Figure 4b shows the intra- and inter-warp hits for the L1 data cache with the infinite size. The figure shows that the average hit rate is about 96%, which is close to a typical hit rate for a CPU L1 cache. Concretely, on an average, we have 50% intra-warp and 46% inter-warp hits. There are a couple of interesting observations. First, kernels actually have a significantly higher locality but the current GPUs are not able to fully exploit it. Second, in contrast to some existing work [15], which shows much higher intra-warp locality than inter-warp locality, we see that there is almost equal division between the intra-warp and inter-warp localities. This is possible for memory-divergent workloads as they have scattered memory accesses and threads from different warps (which contributes to inter-warp hits) can access the data from the same cache line. This can happen more frequently for memory-divergent workloads compared to regular workloads.

We also study the intra- and inter-warp locality for the L2 data cache. Figure 5 shows the intra- and inter-warp locality for the L2 data cache. The average intra-warp and inter-warp hits are 30% and 15%, respectively, for the 128KB size, while for the infinite size, the average intra-warp and inter-warp hits are 65% and 21%, respectively.



(a) Intra- and inter-CTA locality of L2 data cache with 128KB size.



(b) Intra- and inter-CTA locality of L2 data cache with an infinite size.

Fig. 7: Intra- and inter-CTA locality of L2 data cache.

An important point that is not evident after looking at the intra- and inter-warp locality is whether the warps belong to the same CTA or different CTAs (co-operative thread array aka thread block). This could also provide useful insight as CTAs are units of work distribution across different streaming multiprocessors. Moreover, the quantitative numbers further provide information on making scheduling decisions, for example, should we first schedule the warps from the same CTA or different CTAs.

4.4 Intra- and Inter-CTA Locality

Figure 6 shows the intra- and inter-CTA locality of the L1 data cache for 16KB and infinite sizes. Figure 6a shows that on an average, there are 68% intra-CTA hits and 5% inter-CTA hits, which shows that a high percentage of inter-warp hits shown in Figure 4 are basically from warps within the same CTA. This is more evident when we look at the average numbers for the infinite L1 data cache. On average, 91% hits are intra-CTA and only 5% are inter-CTA. Figure 7 shows a similar analysis for the L2 data cache. For the infinite L2 data cache, inter-CTA hits are 13%, which shows that there is a potential for better scheduling. For

example, if we can schedule such CTAs on the same streaming multiprocessor, there is a possibility to exploit more inter-CTA locality at the L1 data cache that can lead to better performance.

5 Related Work

Related Work on Locality in GPU Caches: There are several works which report that hardware-managed caches play an important role in the higher performance of GPUs [18,6,3]. It is also well-known that exploiting locality is crucial in GPUs, otherwise, hardware-managed caches can also cause negative performance [7,14]. As sometimes, the use of caches can degrade performance, there have been several studies, which show that bypassing caches could lead to higher performance [9,5,19,11]. However, to the best of our knowledge, there is not much work done to quantify the locality in GPU caches. Rogers et al. [15] propose a cache aware warp scheduling mechanism, based on estimated intra-warp locality, to capture locality that is lost by other schedulers due to excessive contention for cache capacity. They reported that the majority of data reuse observed in highly cache sensitive benchmarks comes from intra-warp locality and therefore, give priority to intra-warp instructions to access L1 data cache. Li et al. [10] quantify the percentage of the inter-CTA reuse, which is based on the data reuse of all the memory requests generated from streaming multiprocessors before they enter L1 cache. In contrast, we present a comprehensive quantitative study of L1 and L2 data caches in GPUs, highlighting the gap between the locality exploited and the maximum locality actually present in workloads.

Related Work on Memory Divergence: Meng et al. [12] introduce dynamic warp subdivision (DWS), which allows a single warp to occupy more than one slot in the scheduler. Independent scheduling entities allow divergent branch paths to interleave their execution, and allow threads that hit to run ahead. The DWS does not improve memory divergence, but results in improved latency hiding and memory level parallelism. Tarjan et al. [17] propose adaptive slip, which allows a subset of threads from SIMD warps to continue execution while other threads in the same warp are waiting for memory. This provides benefits when runahead threads prefetch cache lines for lagging threads.

Zhang et al. [20] propose a software solution called G-Streamline to improve both irregular memory references and control flow. G-Streamline eliminates irregularity by enhancing the thread-data mappings on the fly. Sartori and Kumar [16] propose branch and data herding, which exploit error tolerance of applications to reduce memory divergence. Branch and data herding is based on the observation that many GPU applications produce acceptable outputs even if a small number of threads in a SIMD execution unit are forced to go down the wrong control path or are forced to load from an incorrect address. Lal et al. [8] investigate performance bottlenecks in GPUs and show that several workloads have a high degree of memory divergence that can cause low performance.

6 Conclusions

GPUs are high throughput devices and can deliver peak performance in TFLOPs. However, due to several performance bottlenecks, the peak performance is often difficult to achieve. While caches play an important role in the higher performance of GPUs, they can suffer from thrashing, leading to high miss rate, in particular, caches can perform poorly for memory-divergent workloads. As exploiting data locality is crucial for performance, we conduct a quantitative study of data locality in GPU caches, showing the limits of locality for memory-divergent workloads. Our analysis shows that memory-divergent workloads have significantly higher locality, but the current GPUs are not able to fully exploit it. We show that 57% of the cache lines are never re-referenced, however, the limit study shows that only 30% of the cache lines are never re-referenced and 27% are evicted before re-reference. We further find that about 50% of the L1 data cache capacity is wasted due to data over-fetch. On the one hand, the low spatial utilization of cache lines, as exploited by GPUs, justifies the cache design to fetch sectors based on demand. On the other hand, the much higher actual spatial utilization of cache lines shows the lost spatial locality and presents opportunities for further optimizing cache design. We also study the locality at intra- and inter-warp levels and observe that, in contrast to previous studies, there is a significantly higher inter-warp locality at the L1 data cache for memory-divergent workloads, which can influence warp scheduling policies. Similarly, the locality analysis at the CTA level shows 13% inter-CTA hits at the L2 data cache, which shows the potential for better CTA scheduling across multiprocessors. In the future, we plan to use some of the key insights to improve GPU performance.

References

1. Al-Kiswany, S., Gharaibeh, A., Santos-Neto, E., Yuan, G., Ripeanu, M.: StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC (2008)
2. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA Workloads Using a Detailed GPU Simulator. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS (2009)
3. Cederman, D., Chatterjee, B., Tsigas, P.: Understanding the Performance of Concurrent Data Structures on Graphics Processors. In: Proceedings of the 18th International Conference on Parallel Processing, Euro-Par (2012)
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC (2009)
5. Chen, X., Chang, L.W., Rodrigues, C.I., Lv, J., Wang, Z., Hwu, W.M.: Adaptive Cache Management for Energy-Efficient GPU Computing. In: Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture, MICRO (2014)
6. Hong, S., Oguntebi, T., Olukotun, K.: Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In: International Conference on Parallel Architectures and Compilation Techniques, PACT (2011)

7. Jia, W., Shaw, K.A., Martonosi, M.: Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS (2012)
8. Lal, S., Lucas, J., Andersch, M., Alvarez-Mesa, M., Elhossini, A., Juurlink, B.: GPGPU Workload Characteristics and Performance Analysis. In: Proceedings of the 14th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS (2014)
9. Li, A., van den Braak, G.J., Kumar, A., Corporaal, H.: Adaptive and Transparent Cache Bypassing for GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC (2015)
10. Li, A., Song, S.L., Liu, W., Liu, X., Kumar, A., Corporaal, H.: Locality-Aware CTA Clustering for Modern GPUs. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (2017)
11. Li, C., Song, S.L., Dai, H., Sidelnik, A., Hari, S.K.S., Zhou, H.: Locality-Driven Dynamic GPU Cache Bypassing. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS (2015)
12. Meng, J., Tarjan, D., Skadron, K.: Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In: Proceedings of the 37th annual international symposium on Computer architecture, ISCA (2010)
13. NVIDIA: CUDA: Compute Unified Device Architecture (2007), http://developer.nvidia.com/object/gpucomputing.html
14. Reguly, I.Z., Giles, M.: Efficient Sparse Matrix-vector Multiplication on Cache-based GPUs. In: Proceedings of the Innovative Parallel Computing, InPar (2012)
15. Rogers, T.G., O'Connor, M., Aamodt, T.M.: Cache-Conscious Wavefront Scheduling. In: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO (2012)
16. Sartori, J., Kumar, R.: Branch and Data Herding: Reducing Control and Memory Divergence for Error-tolerant GPU Applications. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT (2012)
17. Tarjan, D., Meng, J., Skadron, K.: Increasing Memory Miss Tolerance for SIMD Cores. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC (2009)
18. Xiao, S., Lin, H., Feng, W.C.: Accelerating Protein Sequence Search in a Heterogeneous Computing System. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS (2011)
19. Xie, X., Liang, Y., Wang, Y., Sun, G., Wang, T.: Coordinated Static and Dynamic Cache Bypassing for GPUs. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA (2015)
20. Zhang, E.Z., Jiang, Y., Guo, Z., Tian, K., Shen, X.: On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (2011)